TicketCloud

Senior Project

Ross McKelvie
Gene Fisher
Monday, September 16, 2013

TicketCloud is a digital ticket service that offers advantages to both event hosts/vendors and consumers. Using QR Codes, RESTful cloud APIs, and mobile devices TicketCloud takes an outdated market digital and opens up new methods of transacting, allowing customers to resell tickets from their couch and for vendors to reap the benefits of these transactions.

Introduction

Description of the Problem

Ticket buying isn't as easy as it should be. Big name providers such as TicketMaster are slow to embrace new technology. As the rest of the world moves to completely digital transactions and utilizes the Internet for faster and easier transactions, ticket services are behind the curve.

TicketCloud turns this paper-run market digital for vendors and consumers. EventBrite, another online company working to digitalize tickets, does the same. However, since the tickets are digital and tied to a user's account, the tickets are non-transferrable and cannot be returned or cancelled.

TicketCloud will allow users to sell their ticket back in the marketplace for a small fee. This feature is attractive to users of EventBrite and similar platforms that do not allow for the resell of tickets. The ticket will be sold like any other ticket on the platform, and database markers will be updated. The monetary amount returned to this user will be less than the amount originally paid. This fee will be split with the vendor, incentivizing vendors to use the system.

The feature described above is not enabled on any of the ticket services. In regards to physical, traditional tickets, selling a ticket requires an in-person transaction or using a postal service to trade the ticket to the new user. This is time consuming and costs the seller transportation fees and time. Enabling users to resell their ticket from the comfort of their couch is a sought-after feature to users. On the other hand, vendors don't receive any monetary gain from the reselling of tickets, as it happens post transaction between two third parties.

Overview of the Solution

Digitalizing tickets is a simple substitution: remove paper and throw software at the problem. One in ten cellphone owners is using an iPhone. Android is gaining in popularity and is a fierce rival. Utilizing the hardware in everybody's pocket, TicketCloud will efficiently remove the need for paper and integrate seamlessly into patrons' lives.

Fixing this problem with technology will require two sides, a web application that includes a frontend web interface and API endpoints. The API endpoints will be consumed by the iOS applications, one for vendors and one for ticket consumers.

User Levels

There are two user groups, *standard user* and *vendor*. Upon registration, everybody is a standard user but has the option to upgrade their account to vendor. Upgrading

to vendor enables access to the vendor portal (create and manage venues and events) as well as the vendor iPhone app.

QR Codes and Scanning

In order for tickets to be read, a unique QR code is generated for every purchased ticket that contains the link to the user account and the amount purchased, all verified server side. Because events and venues differ, sometimes a paper copy of a ticket will be required. Instead of bringing in a device, users can download and print their tickets. The downloaded PDF contains basic information about the event as well as a large QR code to be scanned by event staff.

Utilizing the iPhone camera, vendors are able to scan QR codes from attendants' paper printouts and mobile devices alike. The camera is able to read the QR code from a digital screen as quickly and accurately as paper.

Web App

The web app is a standard interactive website where users can browse and purchase tickets. Users are able to upgrade their account to **vendor** level, where the following abilities are enabled:

- Ability to create Venues
- Ability to create Events
- Access to the Vendor iPhone app

API

In order for the iOS applications to interface with the database, a RESTful API with relevant endpoints is utilized. The API accepts HTTP GET, PUT, POST, and DELETE packets and returns JSON data to the client.

iPhone: Vendor Application

The first iPhone app is a vendor app, where vendors can sign in to start scanning tickets for an event. The app contains a list of available events. Clicking into an event allows the vendor user to start scanning tickets and marking them as used as patrons enter the event.

iPhone: Consumer Application

The second iPhone app is the consumer application. Within this application, consumers can browse and purchase tickets, as well as pull up tickets to be scanned at the venue. The functionality of browsing and purchasing tickets is identical to that of the web application, minus the vendor functionality.

Scope and Future Work

Payment Processing

For the scope of the project and ease of initial customer development, payment processing is not included. Prior to TIcketCloud's launch, a payment processing API

such as Stripe, Square, or Balanced would need to be implemented, as well as transaction history and receipts for users.

Android App

The apps created for iPhone are native and will not run on the similarly popular Android platform. The demographics of android adoptions are significant enough to warrant android applications in addition to iPhone applications.

Deploy

Taking the product from completed development stage to market is a large process. The following is a list of action items that would need to be taken before launch:

- Corporate entity set up
- Production server environment
- Applications reviewed and accepted into relevant app stores
- Marketing
- Vendor acquisition

Business Plan

The business plan is in an underdeveloped stage and needs to be expanded and researched prior to launch.

Outline of Report

The following report will go in depth into all of the aspects touched upon in the introduction. The Scenario of System Use section contains screen captures of product, walkthroughs of various tasks, and a comparison of functionality between web and mobile applications. The Requirements Gathering section describes the predevelopment process and user polling throughout.

Requirements Gathering

Throughout the initial development process, a variety of requirements gathering techniques were used, ranging from digital mockups to task oriented surveys.

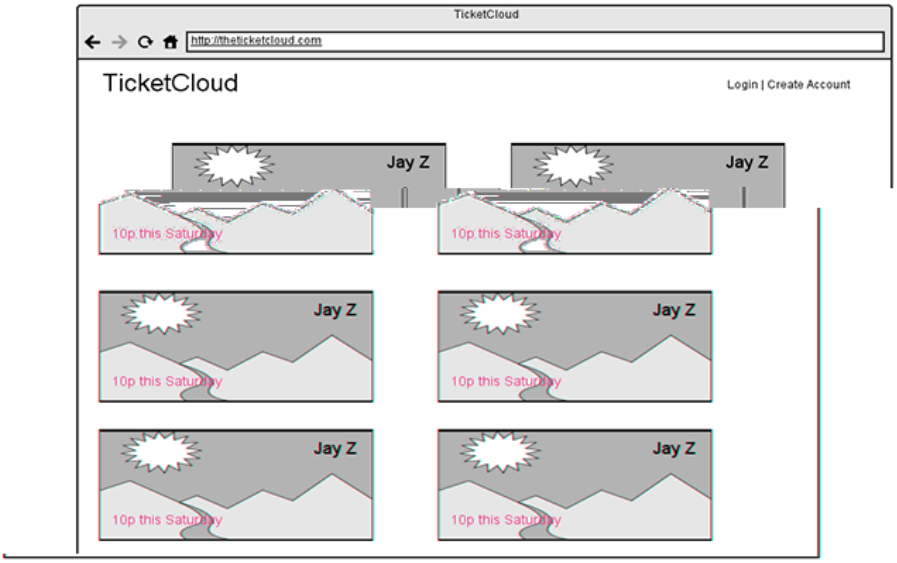
Mockups

Prior to initial development and design, I used a web application called Moqups to design page and app layouts. During this time, I also worked on the flow of the platform, and included notes between wireframes.

The combined wireframes and descriptions follow.

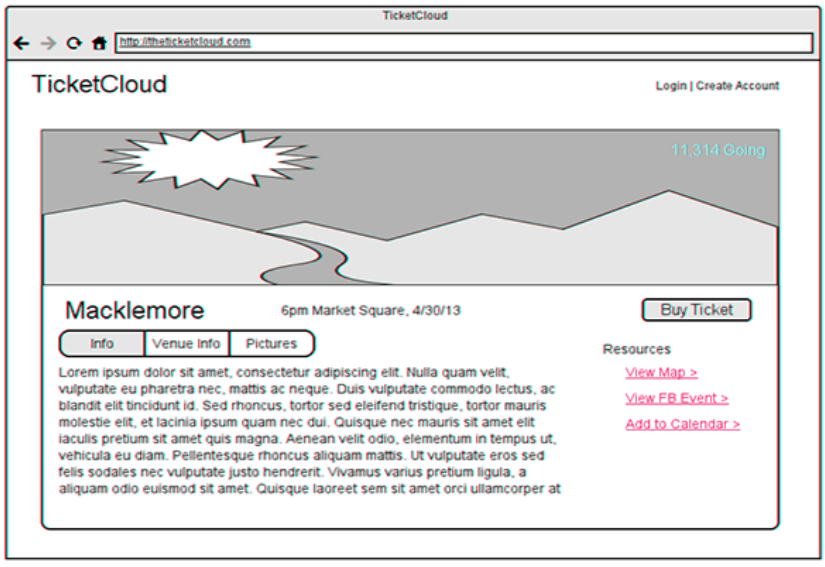
TicketCloud Wireframe & Flow

Users, registered or otherwise, are presented with a grid of events similar to that on the iOS application. Selecting an event takes you to the event details page, where users can buy and sell tickets.



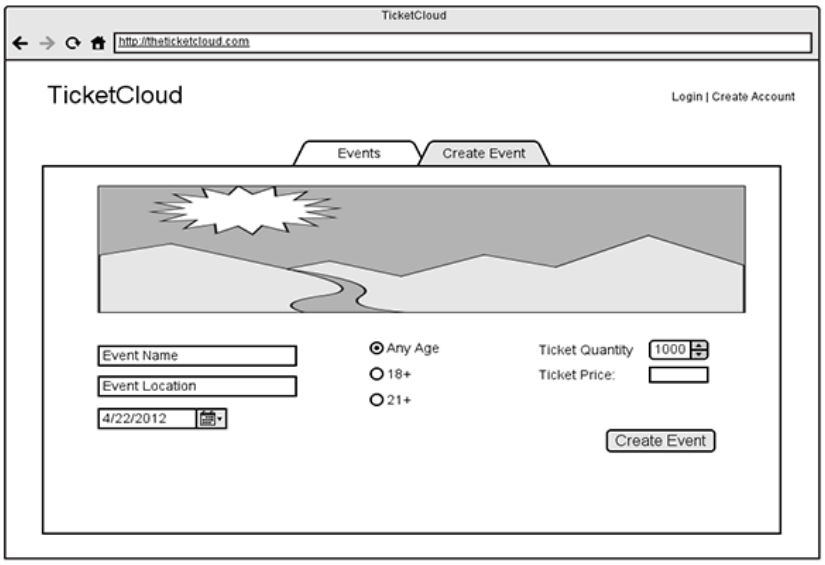
Clicking on an event takes you to the event details page. The top section of the page contains information about the event (# attendees, event name, event time, event place) and a Ticket Purchase/sell button

The bottom area is separated into three tabs: info, venue info, and pictures. The right column contains resource list of: view map, view event on facebook, add to calendar.



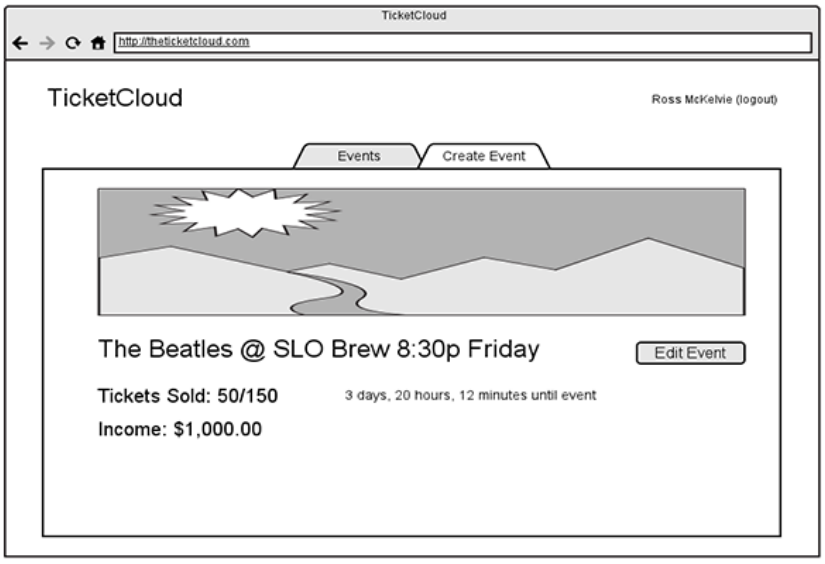
TicketCloud Wireframe & Flow

After creating a Vendor Account, the vendor can create a new event. This consists of dragging a cover photo onto the browser and filling in the event form.



Once the event is created, the tickets are put up for purchase!

The events tab at the top of the vendor website takes the vendor to a list of events they've created. Clicking into an event lets them see metrics and information about the event. Vendors can also click the 'Edit Event' button to make changes to the event.



TicketCloud Wireframe & Flow

A user who opens the app is presented with a grid of upcoming events to select from.

Tapping on any tile takes the user to the Event view, where they see the name and time on top of a banner image (facebook cover photo style). Thumbnails of the event's performers, event descriptions, and a "Buy Ticket" button are also in this view.

If a user has already purchased the ticket, the button will say Sell Ticket, and can place their ticket back into the marketplace after agreeing to a confirmation dialog.

A button on the bottom tab bar links to "My TicketCloud" where the user can see all of the events purchased, and can tap on an event to see more details or sell the ticket (see last frame).

Pilot Survey

In addition to drawing up wireframes during the first week of the project, a pilot survey was also conducted. The survey, consisting of four questions, polled 45 participants about digital tickets and purchases.

The first three questions asked the user to rate their response on an importance scale, ranging from: Non-important, Somewhat non-important, Somewhat important, Important, Highly important. These questions were:

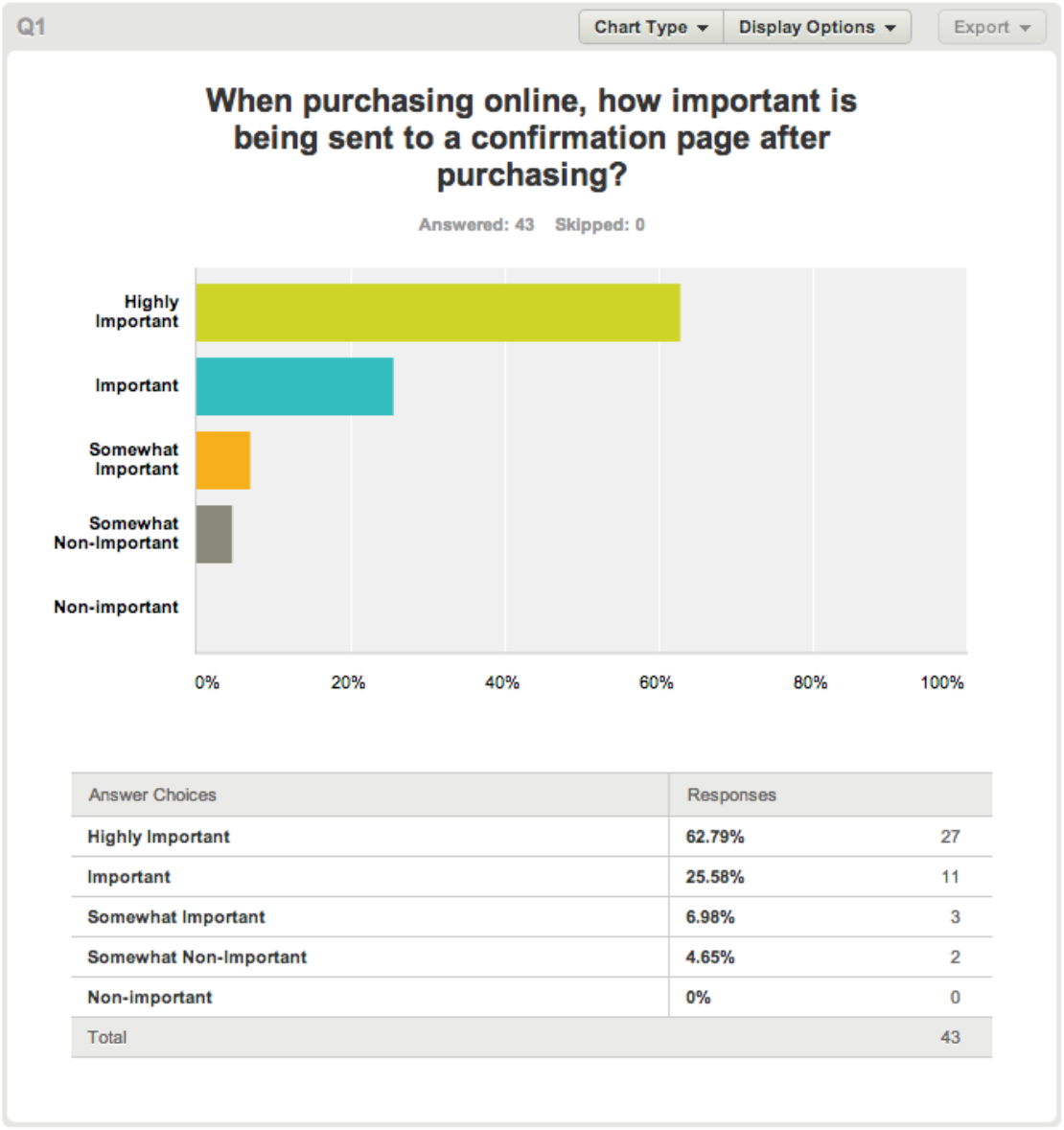
1. When purchasing online, how important is being sent to a confirmation page after purchasing?
2. When purchasing tickets online, how important is being taken to a confirmation page after purchase?
3. When making digital purchases online, how important is being taken to a confirmation page post-purchase?

These questions were asked as I had initially designed the purchase button to change state after purchase, sans a completely different purchase page. These three questions are all very similar, with minor differences.

The fourth question was a simple Yes/No question, with a freeform input box for users answering "Maybe." It asked: would you purchase digital tickets (no hard copy)?

The results of this survey follow. It was very interesting to observe that the majority of people would have no problem with digital tickets, considering the collector and personal value of old concert tickets for attended events.

In addition, 62.79% of participants ranked seeing a standalone confirmation page after making a purchase as Very Important, with another 25.58% ranking it as important. Only 11% ranked a confirmation page as somewhat important or somewhat non-important. This is important to consider when building out the purchase flow as to meet user experience expectations for paying customers.

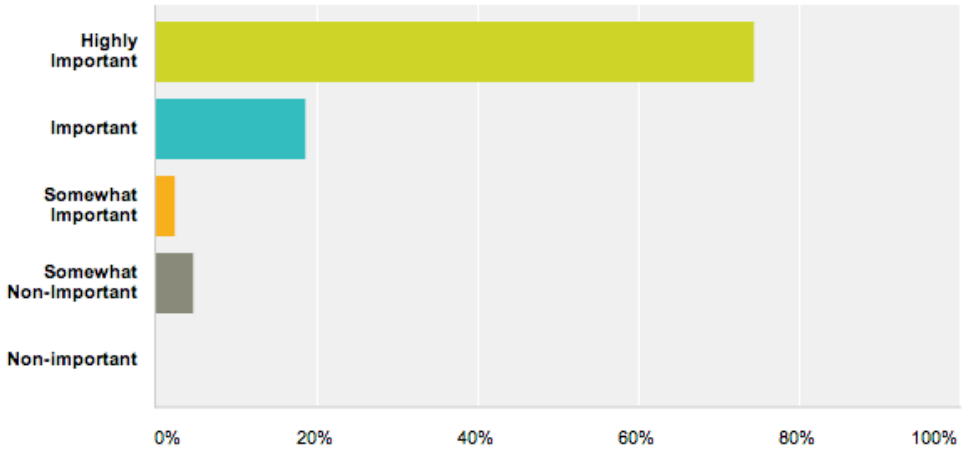


Answer Choices	Responses	
Highly Important	62.79%	27
Important	25.58%	11
Somewhat Important	6.98%	3
Somewhat Non-Important	4.65%	2
Non-Important	0%	0
Total		43

Q2 Chart Type Display Options Export

When purchasing tickets online, how important is being taking to a confirmation page after purchase?

Answered: 43 Skipped: 0

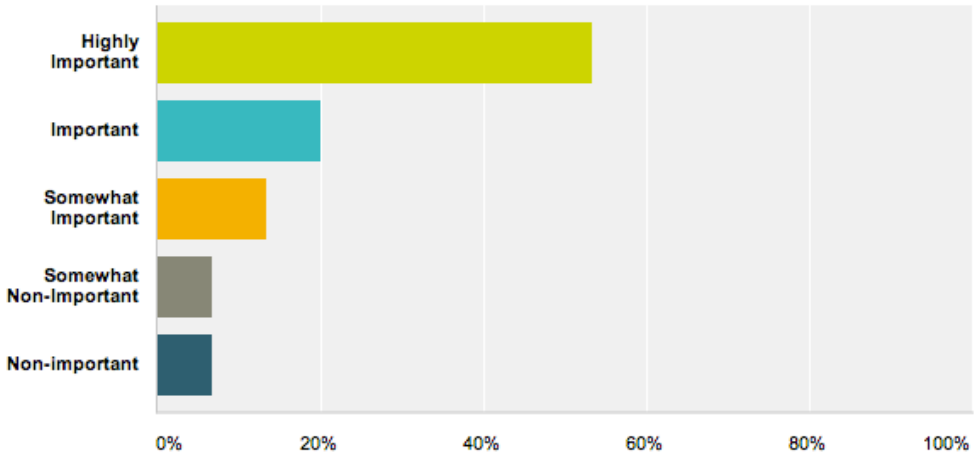


Answer Choices	Responses	
Highly Important	74.42%	32
Important	18.60%	8
Somewhat Important	2.33%	1
Somewhat Non-Important	4.65%	2
Non-Important	0%	0
Total		43

Q3 Chart Type Display Options Export

When making digital purchases online, how important is being taken to a confirmation page post-purchase?

Answered: 45 Skipped: 0

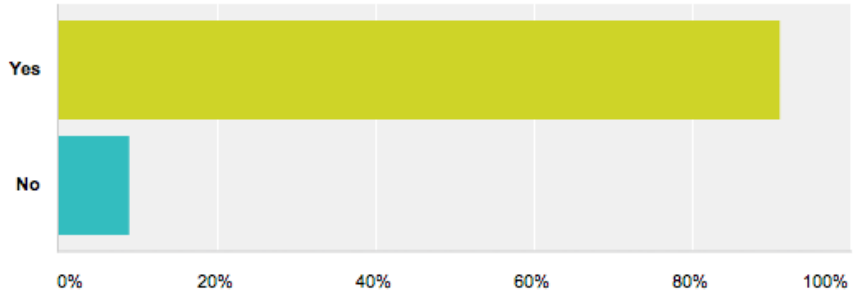


Answer Choices	Responses	
Highly Important	53.33%	24
Important	20%	9
Somewhat Important	13.33%	6
Somewhat Non-Important	6.67%	3
Non-important	6.67%	3
Total		45

Q4 Chart Type Display Options Export

Would you purchase digital tickets (no hard copy)?

Answered: 45 Skipped: 0



Answer Choices	Responses	
Yes	**91.11%**	41
No	**8.89%**	4
Total		45

Maybe (please specify) (6) Hide

Responses (6) Text Analysis My Categories

PRO FEATURE
Use text analysis to search and categorize responses; see frequently-used words and phrases. To use Text Analysis, upgrade to a GOLD or PLATINUM plan.

Upgrade Learn more »

Categorize as... Filter by Category Search responses

Showing 6 responses

after checking reliability of website
2/5/2013 5:16 PM View respondent's answers

sup kert
2/4/2013 5:31 PM View respondent's answers

Confirmation needed
2/4/2013 5:20 PM View respondent's answers

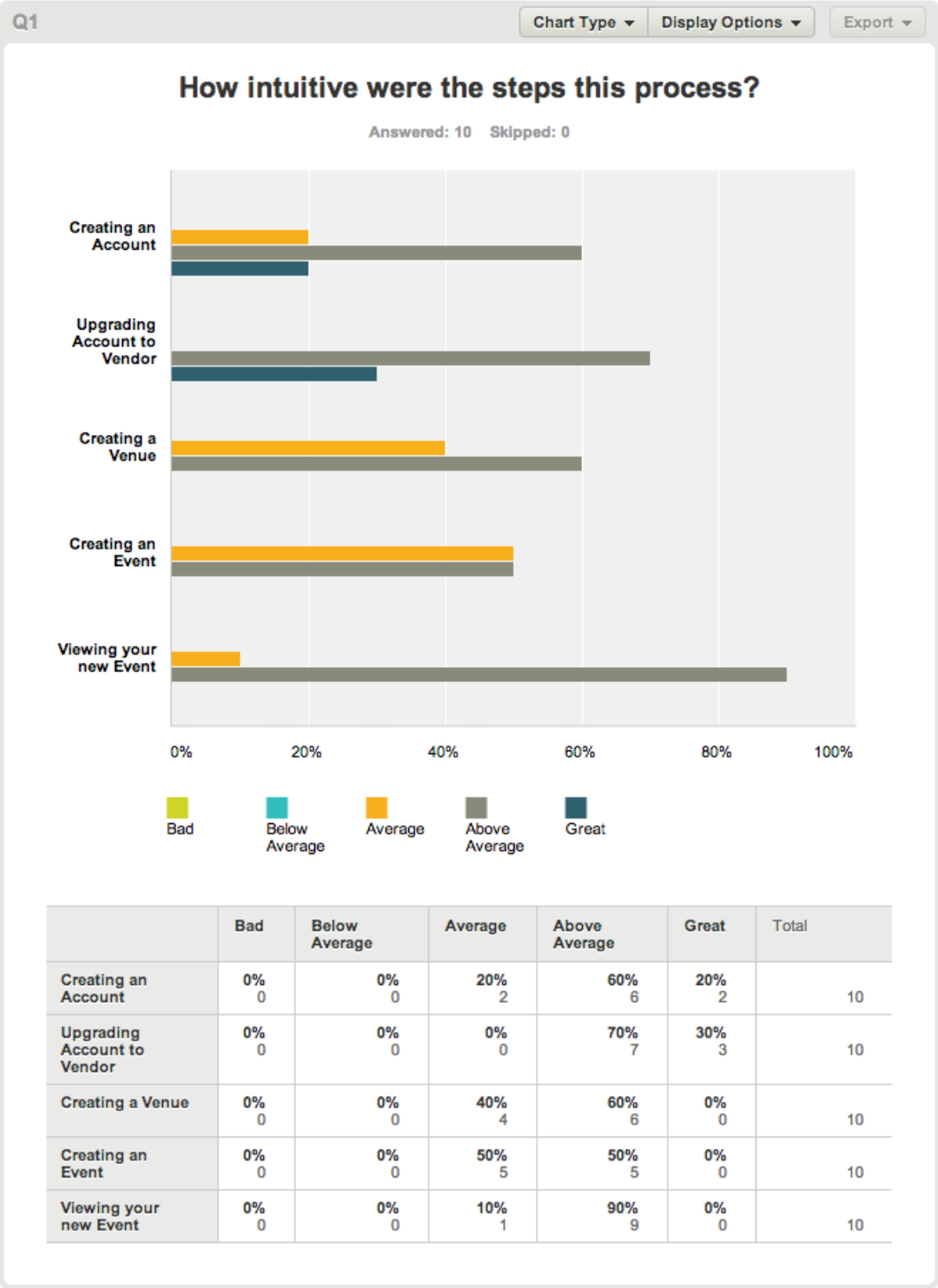
If this means I am not allowed to print something out and take it, No I wouldn't buy digital. I don't trust the people to not lose my ticket. I'd rather have something in my hand.
2/4/2013 4:29 PM View respondent's answers

Only from a well known or otherwise trustworthy site.
2/4/2013 4:20 PM View respondent's answers

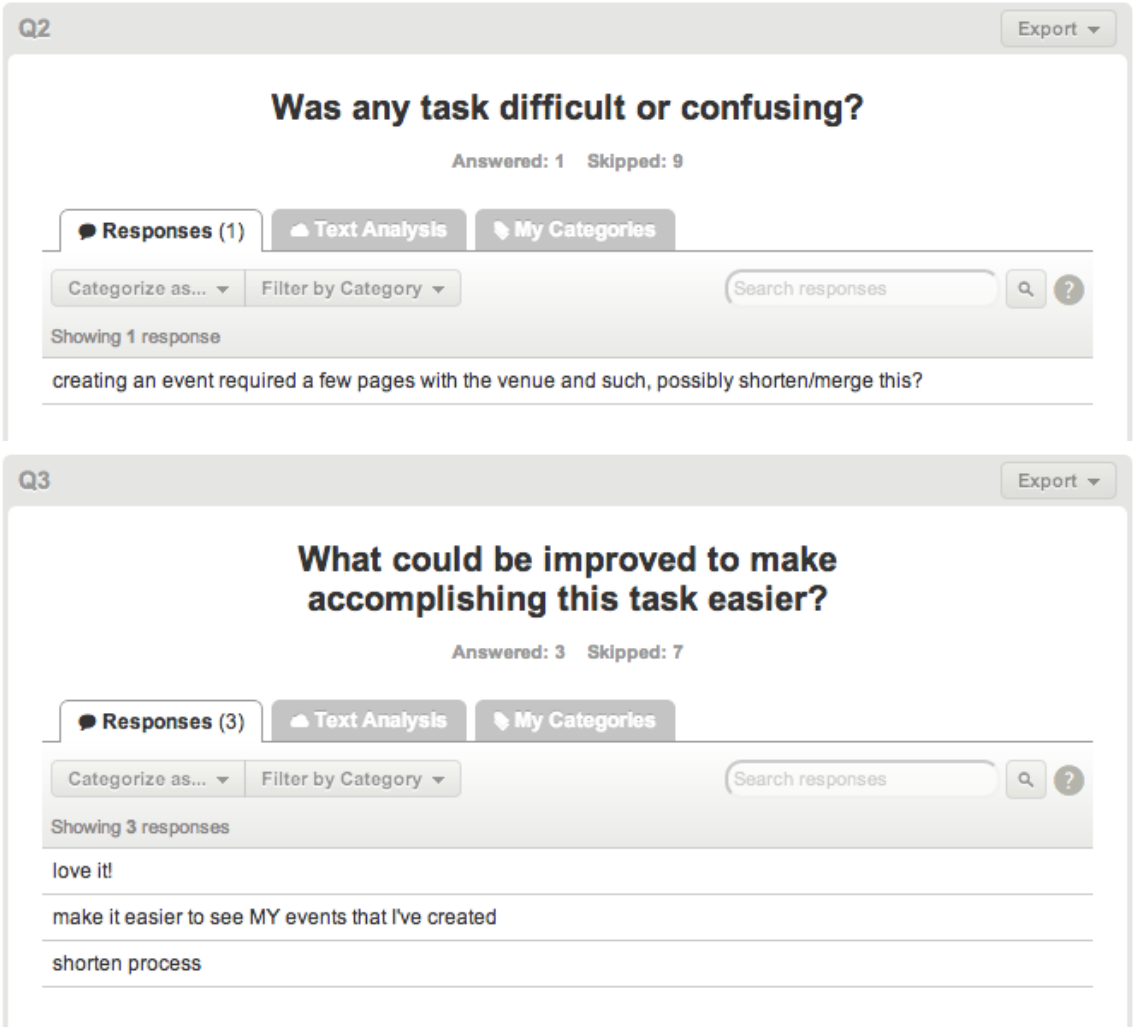
If there was the option for me to print a hard copy. Or proof that I had purchased the digital tickets. I don't think I'll ever have enough faith in technology to let an evening of awesome rest on a database..
2/4/2013 4:20 PM View respondent's answers

Vendor Web App Survey

The vendor web app survey was a task-oriented survey conducted between the first and second senior project quarters. The web application was hosted on a cloud server and the users were given a task to register an account, upgrade it to a vendor, and create an event. The results of the survey follow.



	Bad	Below Average	Average	Above Average	Great	Total
Creating an Account	0% 0	0% 0	20% 2	60% 6	20% 2	10
Upgrading Account to Vendor	0% 0	0% 0	0% 0	70% 7	30% 3	10
Creating a Venue	0% 0	0% 0	40% 4	60% 6	0% 0	10
Creating an Event	0% 0	0% 0	50% 5	50% 5	0% 0	10
Viewing your new Event	0% 0	0% 0	10% 1	90% 9	0% 0	10



The results of this survey brought to light the multiple steps of creating an event. In the future, this multi-page "wizard" will become more of an interactive page that allows vendors to create venues and events from one dashboard workspace. Because vendors will be the revenue generators creating events, this insight is important to creating a user experience that trumps existing competitors.

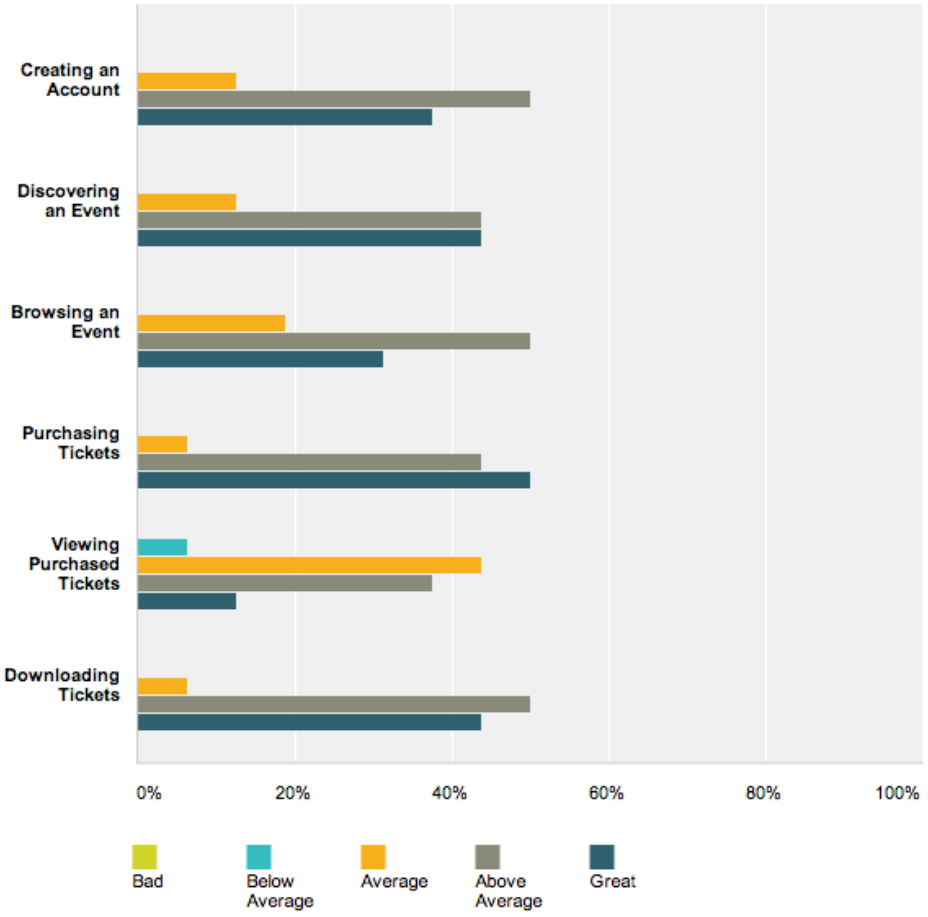
User Web App Survey

Similar to the above survey, but conducted after users had created test events, the user web app survey was a task-oriented survey conducted between the first and second senior project quarters. The web application was hosted on a cloud server and the users were given a task to browse events and "purchase" a ticket. The results of the survey are below.

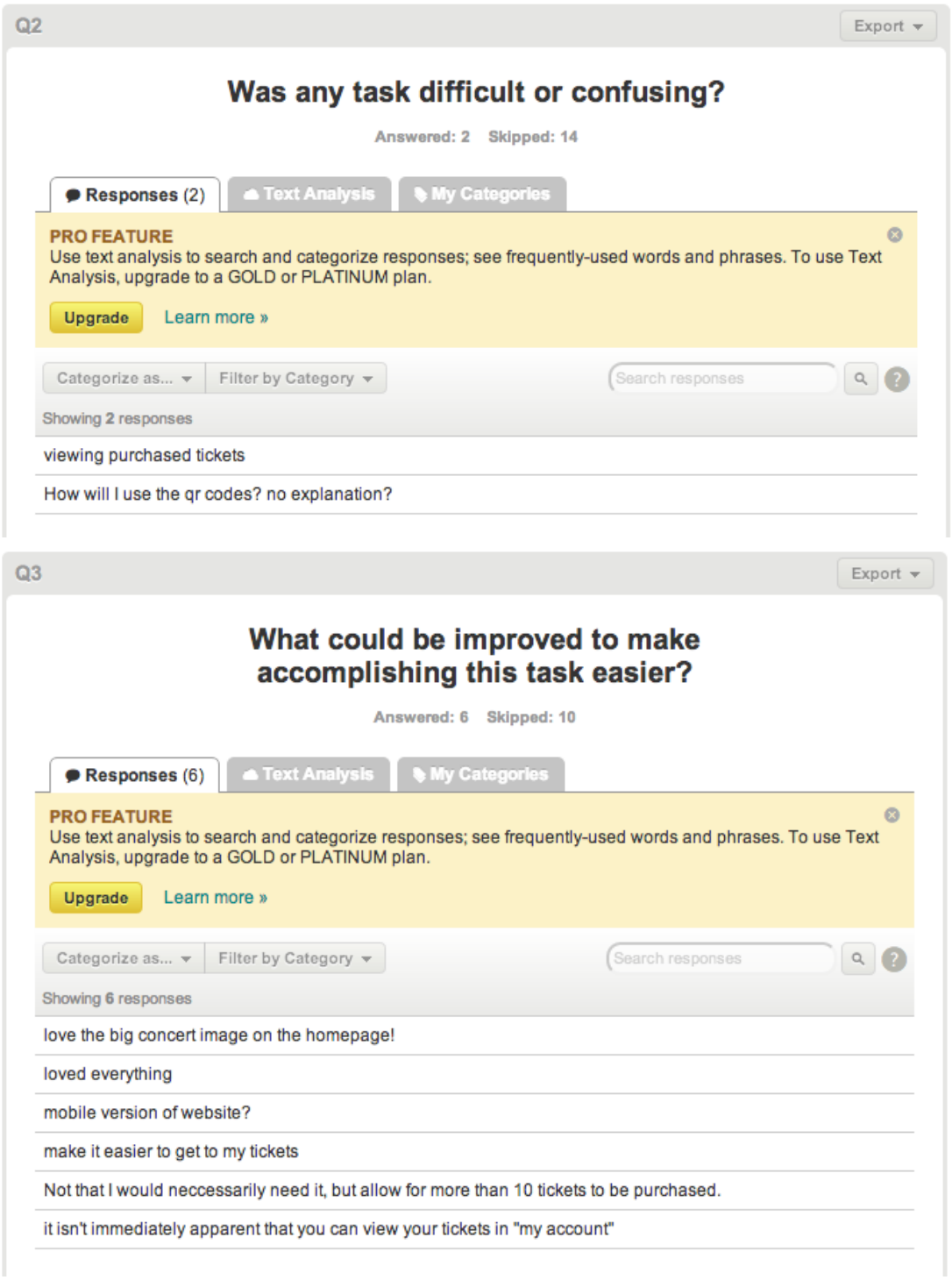
Q1 Chart Type Display Options Export

How intuitive were the steps this process?

Answered: 16 Skipped: 0



	Bad	Below Average	Average	Above Average	Great	Total
Creating an Account	0% 0	0% 0	12.50% 2	50% 8	37.50% 6	16
Discovering an Event	0% 0	0% 0	12.50% 2	43.75% 7	43.75% 7	16
Browsing an Event	0% 0	0% 0	18.75% 3	50% 8	31.25% 5	16
Purchasing Tickets	0% 0	0% 0	6.25% 1	43.75% 7	50% 8	16
Viewing Purchased Tickets	0% 0	6.25% 1	43.75% 7	37.50% 6	12.50% 2	16
Downloading Tickets	0% 0	0% 0	6.25% 1	50% 8	43.75% 7	16



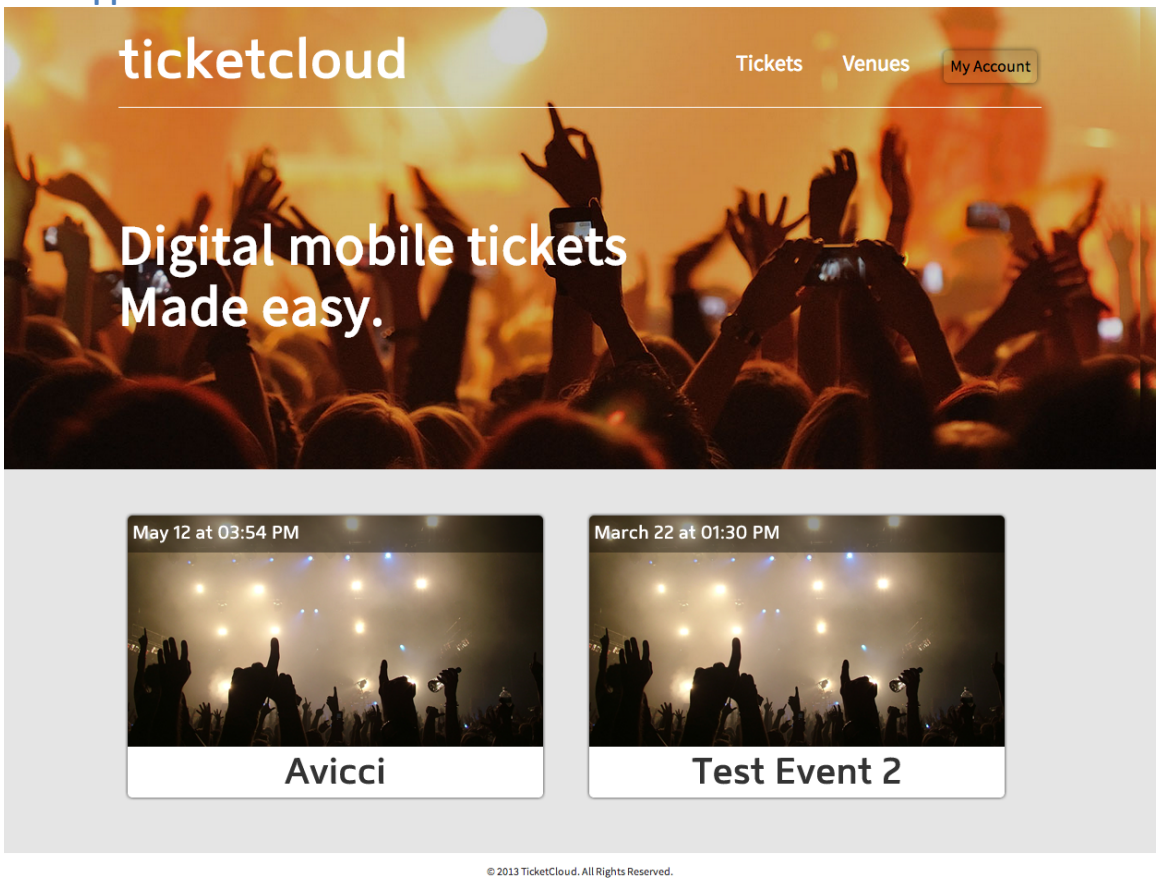
This survey concluded that, at the time, it was difficult to browse one's purchased tickets. This was the highest voted pain point for the task, and most mentioned in the last question regarding improvements. This was great information to receive, and led to interface changes that clearly displays purchased tickets with clear headings and buttons.

Scenario of System Use
The scenario is based on an actual concrete session where the user performs some interesting and useful work. The ultimate goal of this section is to illustrate the major and interesting system features of TicketCloud.

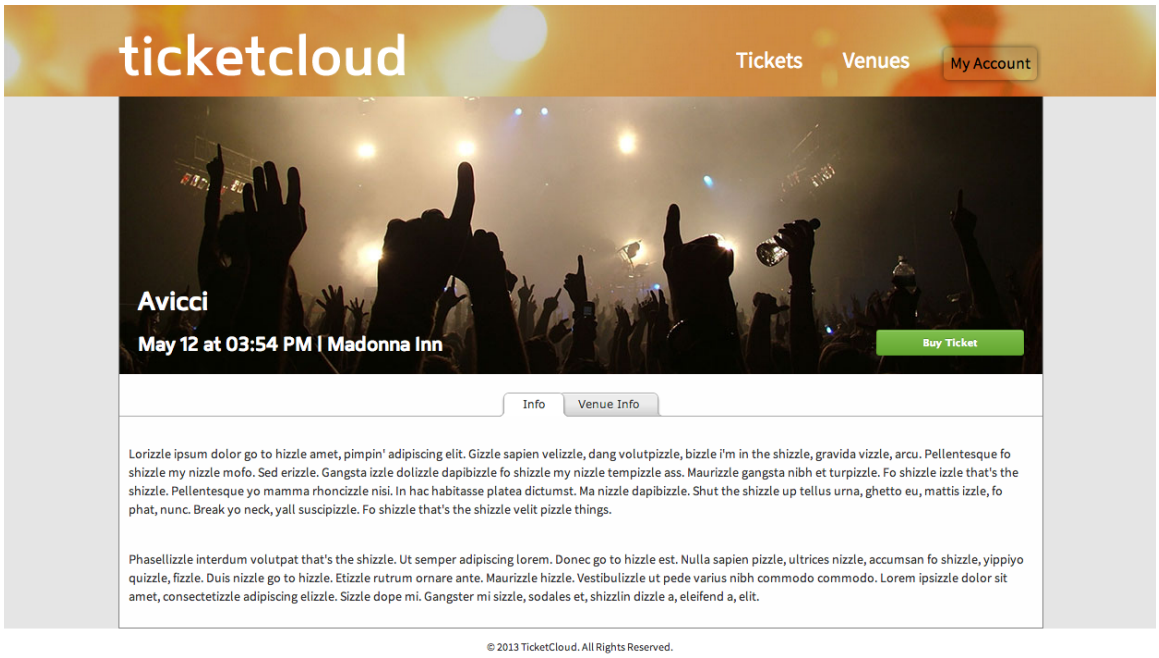
The screenshots below do differ from initial wireframes, but only slightly. The majority of the information was displayed nearly the same, with only minor differences. Such as a one element per row table layout in production with the mockup having a two element per row "grid" layout. That being said, creating mockups was an important exercise that allowed me to get early stage customer feedback, as well as quickly layout my thoughts for the initial user interface & experience designs.

Product Walk-Through

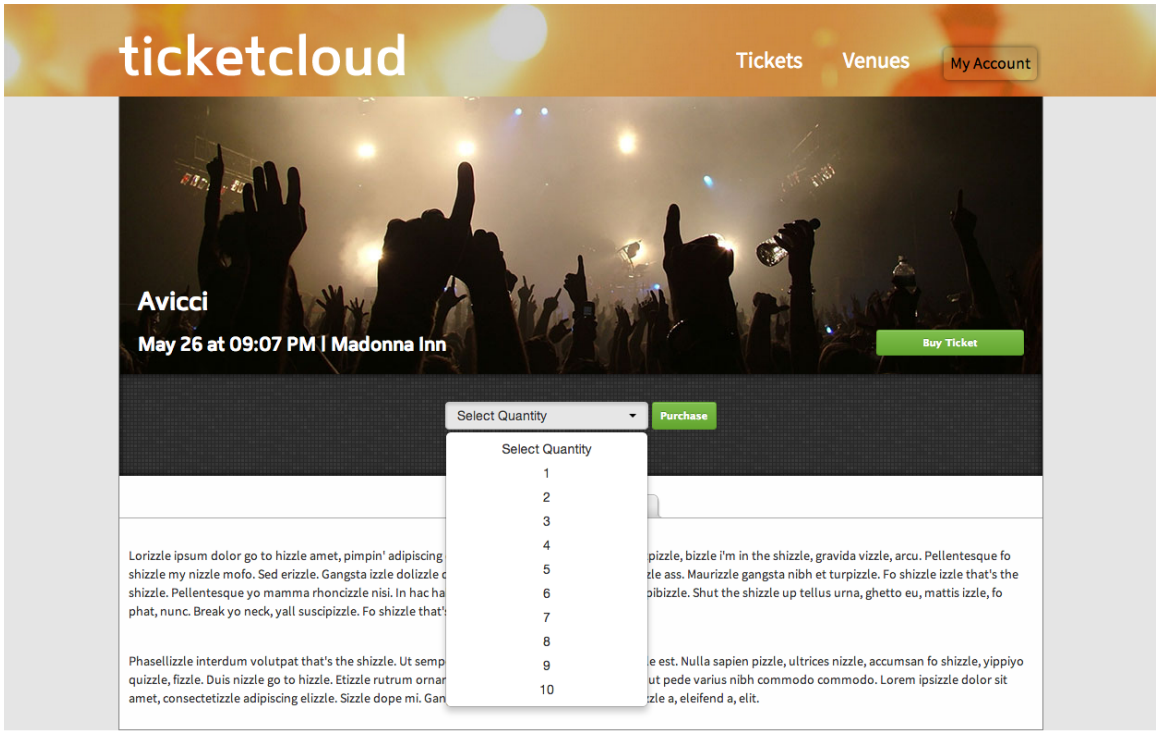
Web App



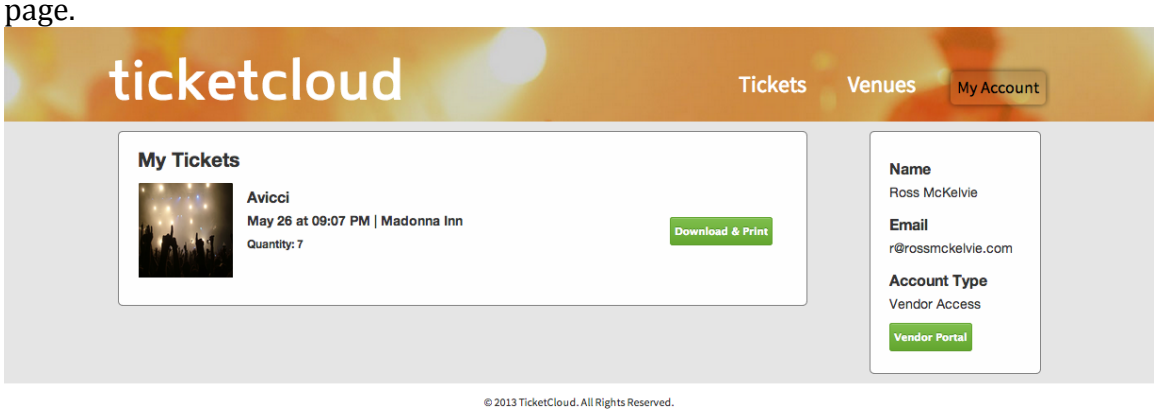
The homepage of TicketCloud includes header links to relevant sections of the website, a clean banner that states the value proposition of TicketCloud, as well as an explore grid containing all events a user can browse.



Selecting an event takes the user to the event details page, where they can see a larger picture, an information tab regarding event information, as well as a Venue Info tab that contains a map, website, and other details about the venue for the event.

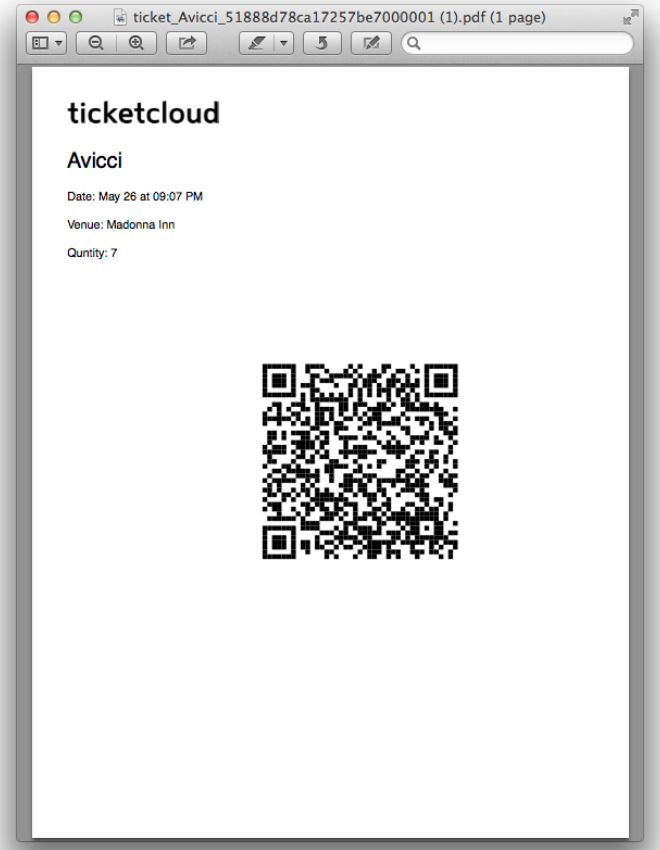


Clicking on the green “Buy Ticket” button drops down the purchase form, where users can select a ticket quantity and finalize the purchase. This slide-down form saves the user from having to navigate away from the details while considering purchasing tickets. After a ticket is purchased, the user is shown a new confirmation page.

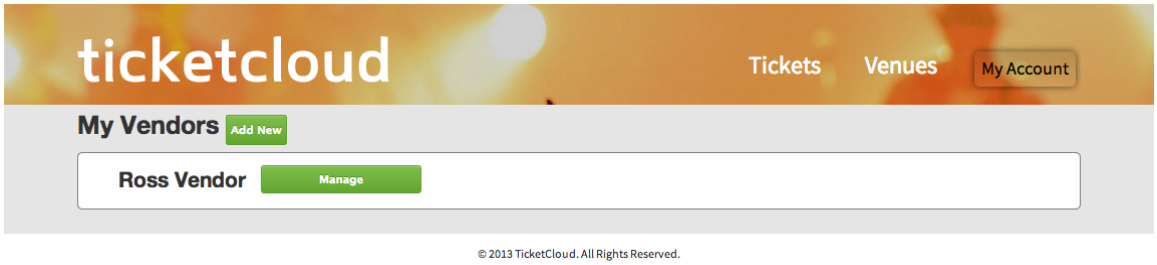


The “My Account” page, accessibly by clicking the button at the top right of every page , contains the user’s basic information and purchased ticket list.

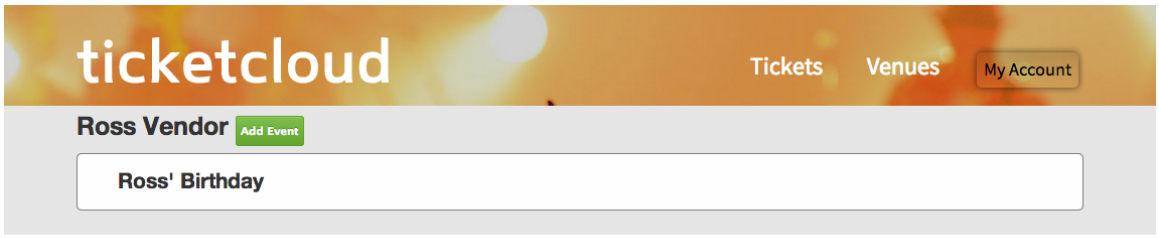
The right side of the view contains the account type, with an action button below where the user can upgrade his or her account to a vendor, **or** access to the vendor portal.



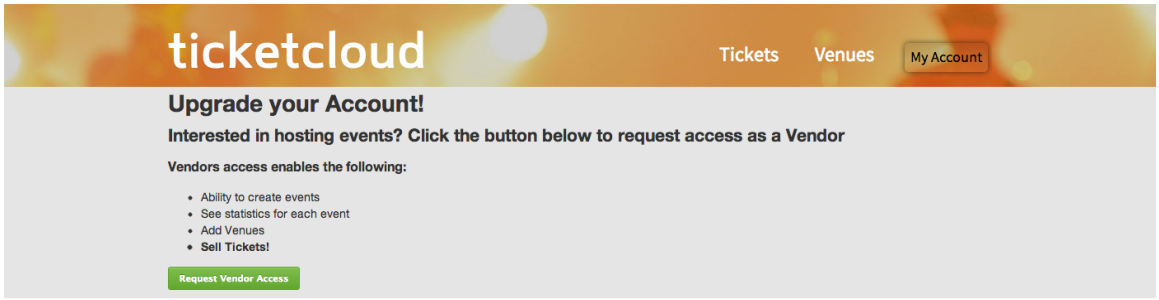
Clicking the “Download & Print” button on a purchased ticket generates a PDF with the corresponding QR code for the user, as well as basic event information specifically the event name, date, venue and ticket quantity.



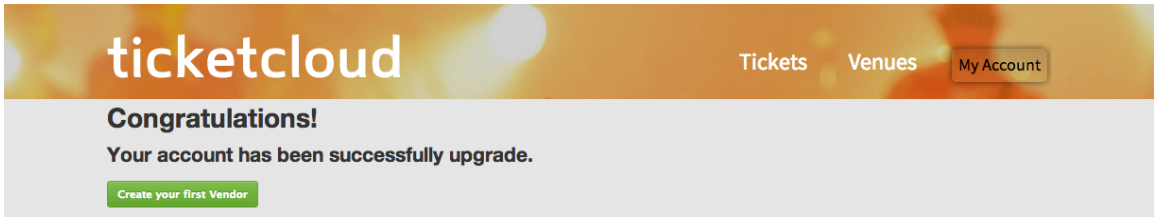
The Vendor Portal contains a list of the user’s vendors, with buttons to create a new vendor, or manage an existing vendor. Selecting a vendor opens an edit form for the vendor.



Managing a vendor contains a list of events for the Vendor, as well as a button to add an event. Selecting an event opens an edit form for the event.

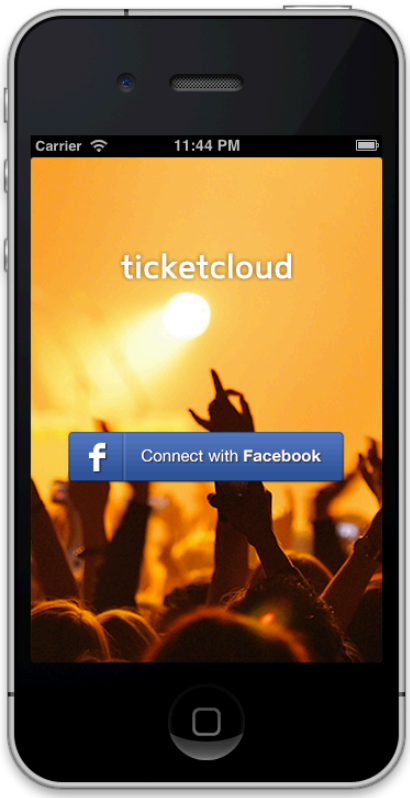


If a user is not a vendor and instead presses the “Upgrade” button, as opposed to the “Vendor Portal” button, on the My Account page, they are brought to a Vendor information page, above.



Once the user requests vendor access, they are given a confirmation. In the future, there may be an approval process, but for the sake of development and user testing, all users are allow vendor access.

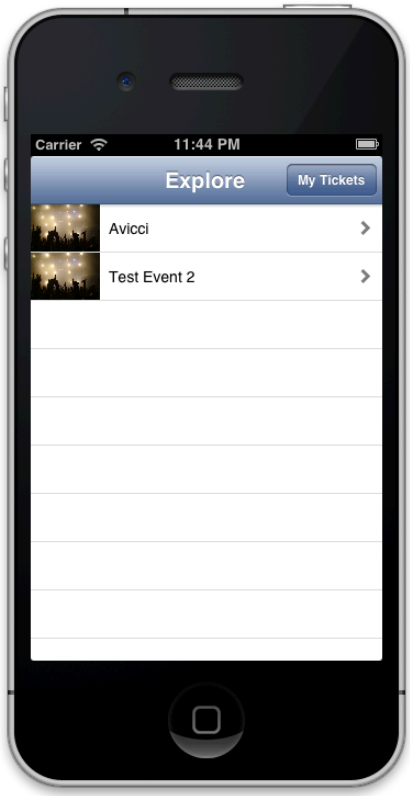
User Mobile iOS Application



First, the user is presented with a splash screen and a button to connect with Facebook. Pressing the Connect with Facebook button will prompt the user to accept access to the application – they do not need to be previously registered on the website.

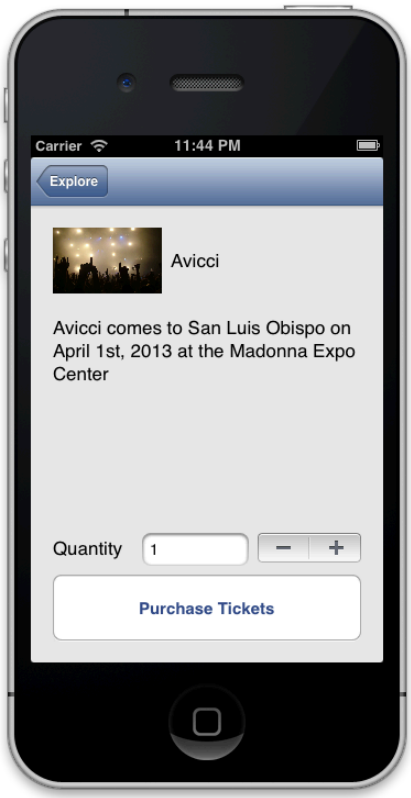
If a user requests login that the server API does not recognize, the Facebook access token is used to fetch user info server side and create a new record for this user, which is then returned the to app.

If an existing user is requesting login, the existing record and corresponding ticket purchases is sent back to the device.



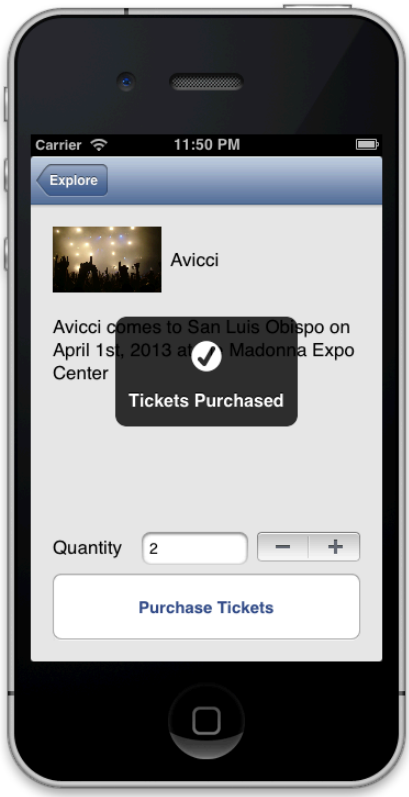
The main screen of the user application is an explore table, containing a list of all available events to the user. Clicking on an event will take the user to a more detailed page about the relative event, and a form for purchasing tickets (see below).

There is also a button in the menu bar, labeled “My Tickets,” that will take the user to a list of their purchased tickets.



To the left is the expanded view for an event, reached by selecting an event in the Explore view. The top half of this view provides the user with the event title, picture, and description.

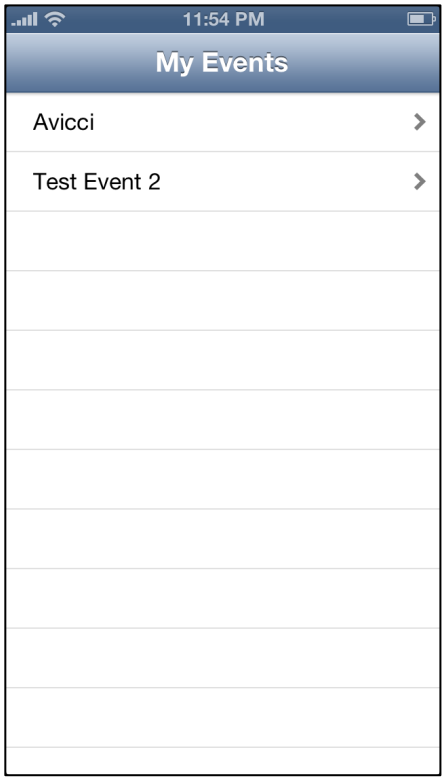
The bottom half of this view provides the user with the ability to purchase tickets. After a ticket is purchased, a success modal is faded in, then faded out after a second, confirming the order.



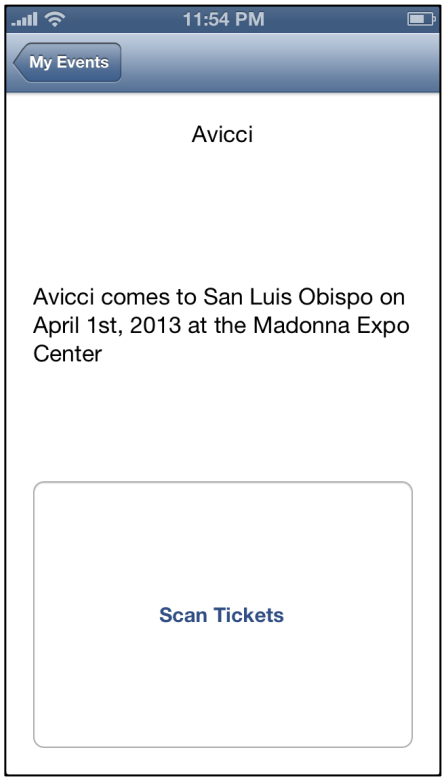
By clicking on the “My Tickets” button from the Explore view, users can see a table of the events they have requested.

Clicking on one of the events will display the title and QR code, to be shown to event staff upon entry. Each ticket can be used only once and is marked as *used* after being scanned by the event vendors.

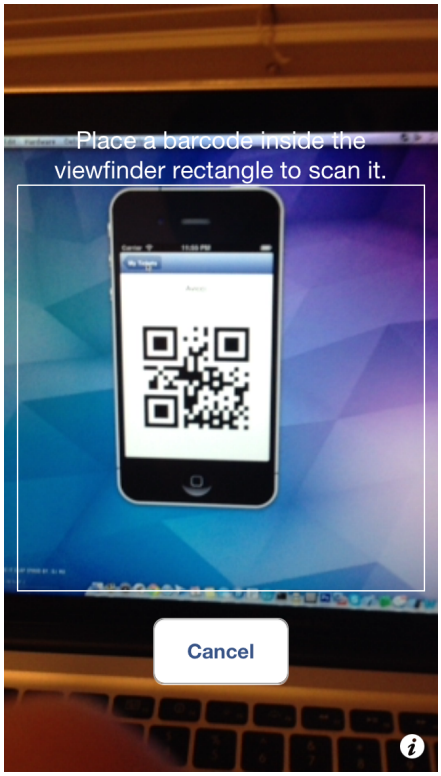
Vendor Mobile iOS Application



After connecting with Facebook, the vendor is presented with the list of their upcoming events.



Selecting an event shows the title, description, and a button for scanning tickets.



Clicking on scanning tickets opens up the QR code scanner and enables the device camera. As the app reads valid tickets' QR codes, the ticket records are marked as *used* via a PUT request to the API.

Web & Mobile Functionality Analysis

Vendor Functionality

A main difference in functionality between the web app and the mobile app falls under vendor access. The web app contains features that allow a user to upgrade his or her account to vendor status, and to create, edit, and delete events.

The vendor-specific mobile app is for use at events. The vendor app contains a QR scanner that scans users' tickets and marks them as used, ensuring that multiple persons do not enter events using the same ticket. The mobile app *does not* allow vendors to create, edit, or delete events, nor does it allow a user to upgrade his or her account to vendor access.

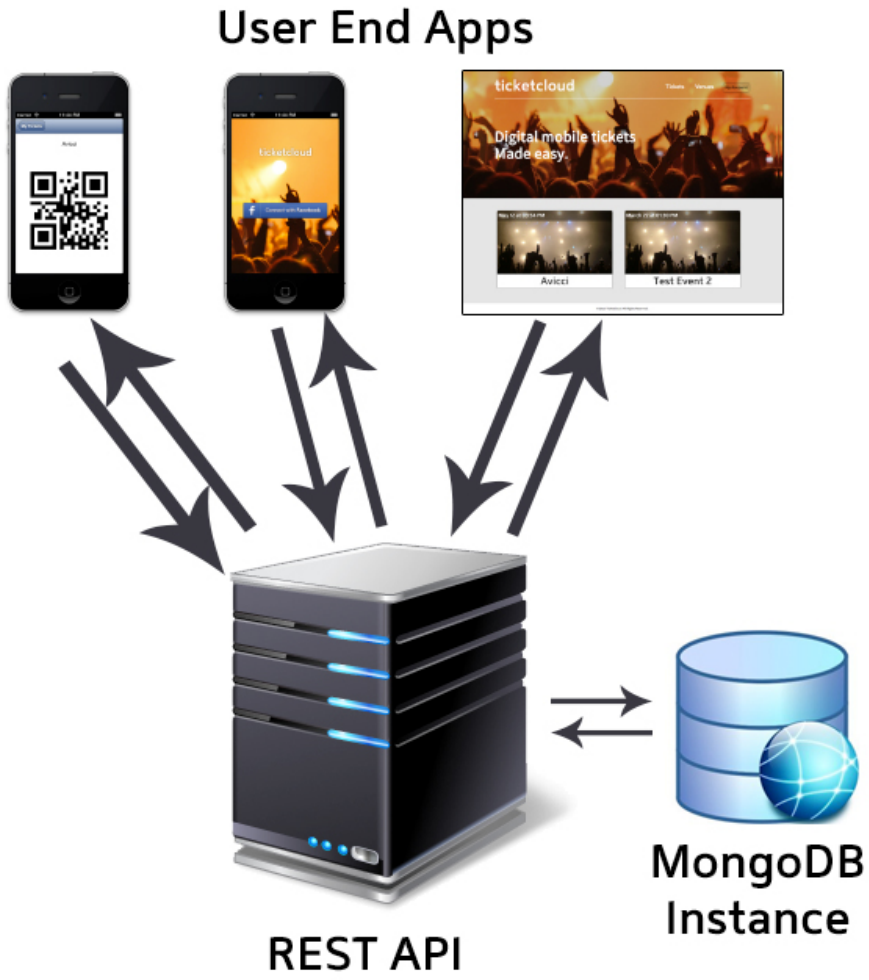
In addition, the web app does not contain any functionality that would allow a vendor to mark tickets as used, like the mobile app can do. Thus, vendors and associates cannot open up the website in a mobile browser and conduct business as they'd expect from the TicketCloud iOS mobile app for vendors.

Design and Implementation of Server & Web Application

Application Level

Overview

Ruby on Rails powers the core server, for both the client-side website and REST API. Defined in the Ruby on Rails (RoR) [7] application are models, shared by both the website and API. The website takes advantage of RoR to generate views, manifest, compile and merge JavaScript, CoffeeScript [1] and SCSS (SASS [10]+ Compass CSS [2]) files, and image sprites. The API takes advantage of RoR's views & JSON templates.



The above high-level diagram displays the flow of information between client-side apps (mobile & web) and the REST API, as well as the flow of data between the API server and the data storage.

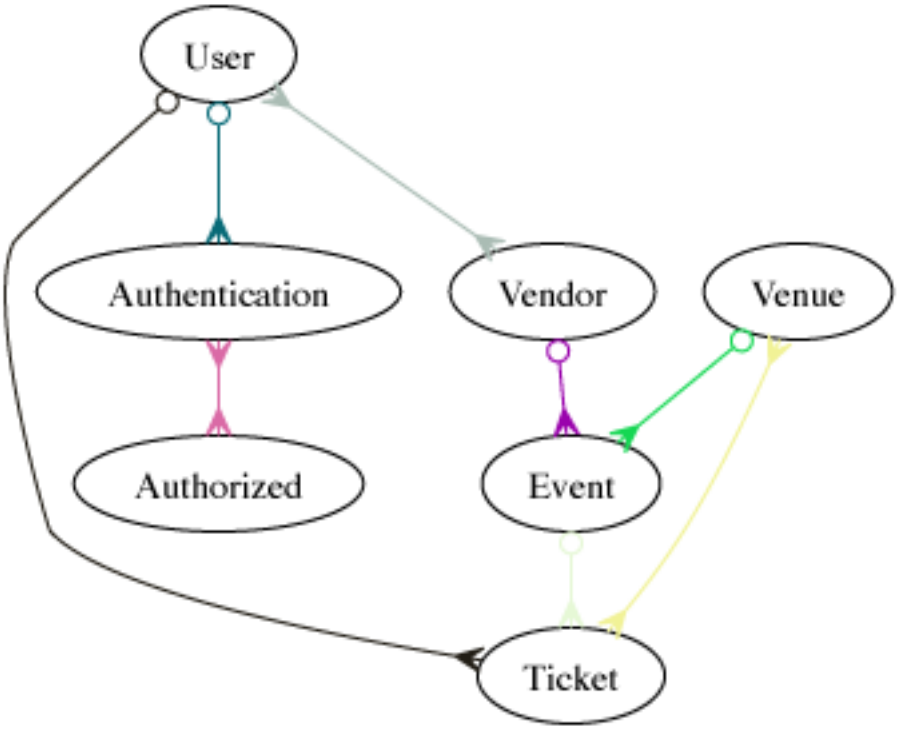
Database

MongoDB [4] is the primary, persistent data storage utilized in the TicketCloud project. MongoDB is a document-based NoSQL database solution, and was picked because of its versatility, document-based optimizations and ease of storage, proven track record, and its ability to have application-level changes not require the database to be taken offline and migrated. A common issue in making later-stage changes in a production environment for a SQL database requires migrating tables before data can be accessed or written, causing downtime. Because MongoDB stores everything as JSON, it can be left up to the application-level code to handle legacy data, and scripted "migrations" can be run at any pace, without resulting in application downtime.

Mongoid [5], an Object-Document-Mapper (ODM) for MongoDB, is the database driver utilized in this project. Mongoid provides a drop-in replacement for the baked-into-rails ActiveRecord for model persistence. Mongoid is responsible for converting rails calls into database queries.

Redis [8], an extremely fast key-value storage, is used for data and view caching. Redis data is stored in memory and thus is faster than MongoDB, so its use cases are for data generated from persisted data. In the case of TicketCloud, it is used primarily for JSON caching on the API end.

Models

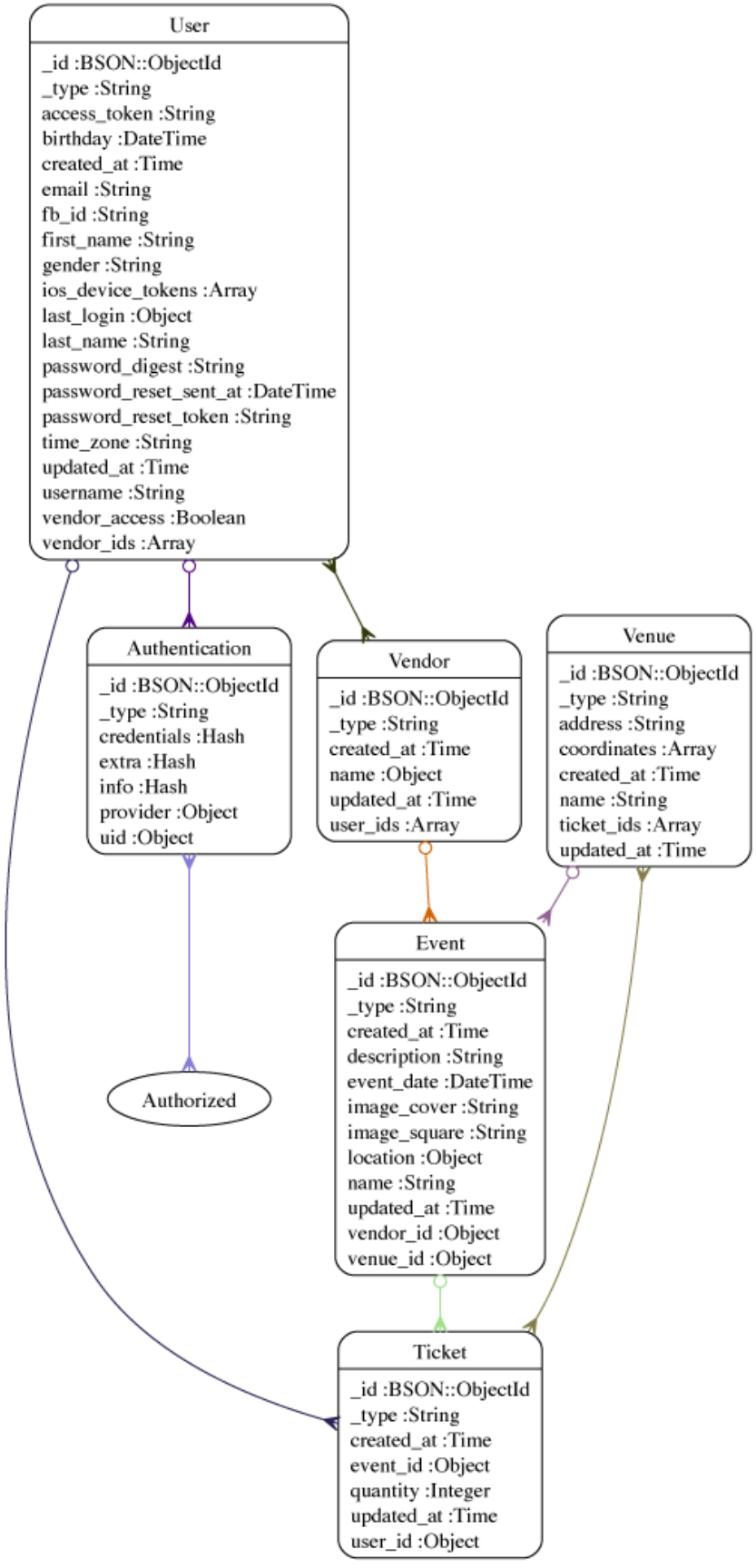


The figure above contains an overview of the application's models and their inter-model relationships.

From a high level perspective, a `User` owns a `Vendor`, which in turn creates `Events` that belong to one `Venue`. A user also contains authentications, these are document-embedded entities that contain authentication information to social networks, such as Facebook. Because the relationship between `User` & `Authentication` is polymorphic, it needs to be given a name. Thus, a `User` is `Authorized` through an `Authentication`.

Users can generate `Tickets`, which have relationships to both the event and the venue. Tickets belong to one user, but multiple tickets can belong to the same user. An event contains many tickets.

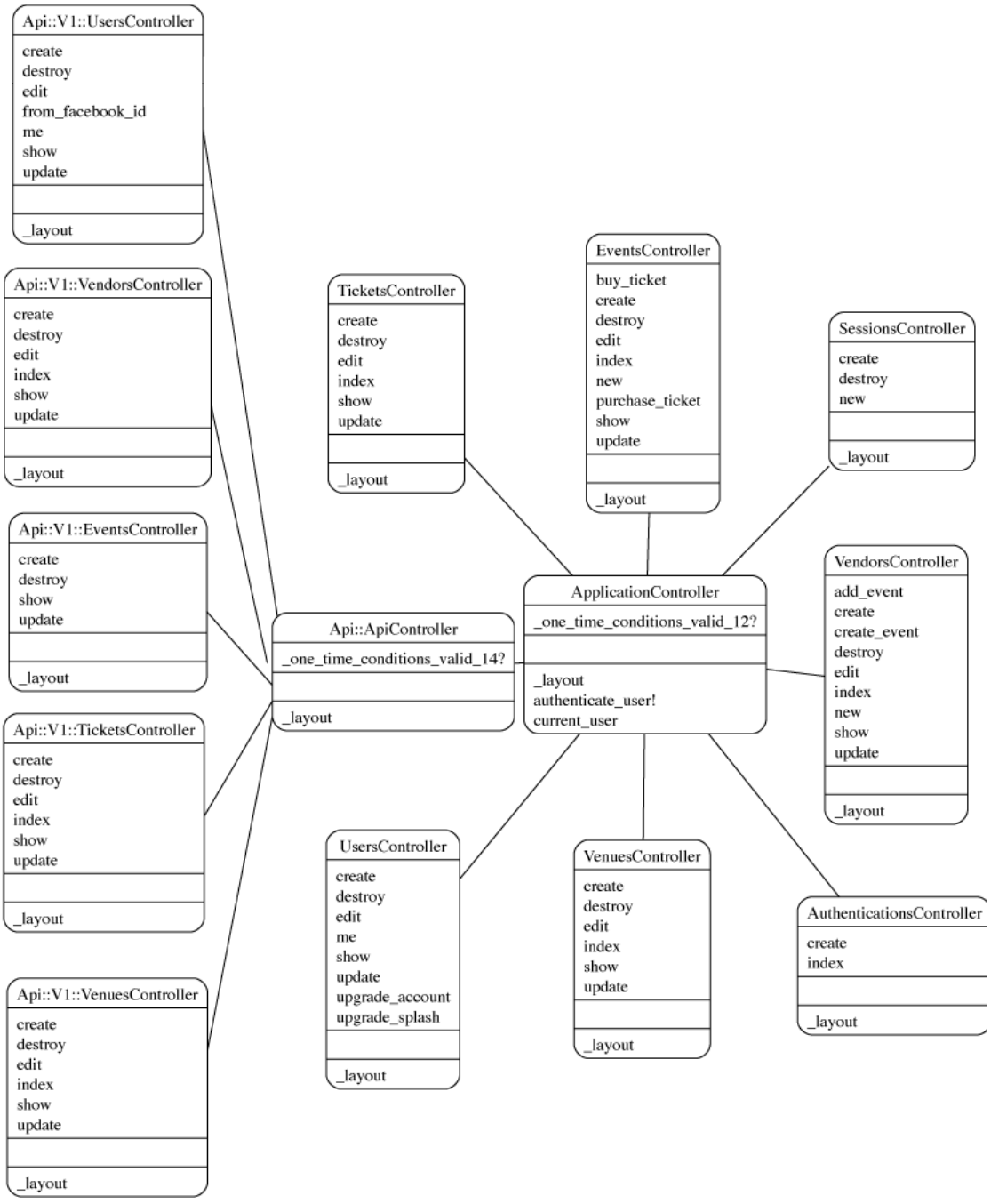
The figure below is an expanded view of the above diagram, containing model attributes in addition to entities and their relationships.


User
_id :BSON::ObjectId
_type :String
access_token :String
birthday :DateTime
created_at :Time
email :String
fb_id :String
first_name :String
gender :String
ios_device_tokens :Array
last_login :Object
last_name :String
password_digest :String
password_reset_sent_at :DateTime
password_reset_token :String
time_zone :String
updated_at :Time
username :String
vendor_access :Boolean
vendor_ids :Array
Authentication
_id :BSON::ObjectId
_type :String
credentials :Hash
extra :Hash
info :Hash
provider :Object
uid :Object
Vendor
_id :BSON::ObjectId
_type :String
created_at :Time
name :Object
updated_at :Time
user_ids :Array
Venue
_id :BSON::ObjectId
_type :String
address :String
coordinates :Array
created_at :Time
name :String
ticket_ids :Array
updated_at :Time
Authorized
Event
_id :BSON::ObjectId
_type :String
created_at :Time
description :String
event_date :DateTime
image_cover :String
image_square :String
location :Object
name :String
updated_at :Time
vendor_id :Object
venue_id :Object
Ticket
_id :BSON::ObjectId
_type :String
created_at :Time
event_id :Object
quantity :Integer
updated_at :Time
user_id :Object

Frontend & API Controllers

The user and API endpoints to the application are diagramed and described below. These endpoints accept and/or display data and content pertinent to their related models, conventionally the word preceding “Controller” for each of the classes.

The diagram below displays the controllers and their inheritance, along with the methods that each provides, typically a set of expected CRUD functions with various extensions.



API Controllers

The API, by RESTful nature, is stateless. Because of this, a user must authenticate and receive an `access_token` that allows that user to process subsequent requests. In order to validate the user is who they say they are, and because the application uses the Facebook platform to authenticate the user, the first API endpoint that must be hit is `Api::V1::UsersController#from_facebook_id` and pass in the Facebook access token. The API can then make a request to the Facebook Graph API and compare the stored information with data returned from Facebook. From here, an access token is generated and associated with the user's account and returned in the JSON request.

Every subsequent request to the API requires that `access_token` be passed in as a parameter, whether in POST data or within the URL parameters. The Ruby/Rails code for validating a user by `access_token` is below:

```
  def current_user
    if params[:access_token] && params[:fb_id]
      @current_user = User.where(
                        :fb_id => params[:fb_id],
                        :access_token => params[:access_token]
                      ).first

      # Unset user if access_token is invalid
      if @current_user.nil? || @current_user.access_token_expires < Time.now
        @current_user = nil
      end

    elsif params[:fb_id] != nil
      @current_user = User.where(:fb_id => params[:fb_id]).first
      @current_user =
User.create_new_user_from_facebook(params[:fb_access_token]) unless
@current_user
      handle = Koala::Facebook::API.new(params[:fb_access_token])
      data = handle.get_object('me')

      # Make sure we're the same facebook user
      unless data['id'] == params[:fb_id]
        @current_user = nil
      end
    end

    if @current_user.nil?
      raise InvalidAccessToken
    end

    return @current_user
  end
```

The API only supports JSON as the return data-type. XML could be easily supported, however for my purposes and following industry standards, JSON is preferred.

Frontend Controllers

The frontend controllers are fairly straightforward and conventional. A URL request comes in from a user navigating via a browser, it is routed to the appropriate controller and action where relevant data is retrieved from data stores, and an HTML view is rendered with this data.

For each HTML view, the data is templated using HAML [3], a HTML abstraction markup language. An example of a HTML template with Rails-supplied data is below:

```
#event
  .cover
    %h1
      = @event.name
    %h2
      #{@event.event_date.to_s(:event_time)} | #{@event.venue.name}

  .event-data
    %h2 Purchasing new ticket

    = form_tag "/events/#{@event.id}/purchase" do
      = label :ticket, :quantity
      = select :ticket, :quantity, 1..10
      = submit_tag "Purchase"
```

QR Codes & PDF Tickets

QR codes are dynamically generated at render-time for various views. For instance, when a PDF version of a ticket is requested, the process is the following:

- Query for relevant data from database
- Generate a QR code
- Create a new PDF document
- Place and position relevant ticket data (event name, time, location, etc)
- Place and position QR code

Generation of a QR code is quite simple. QR codes are just a two dimensional, computer-readable representation of arbitrary data. Generated QR codes consist of a text-string that is the underscore-concatenated string of the owner's unique user id and the unique ticket id.

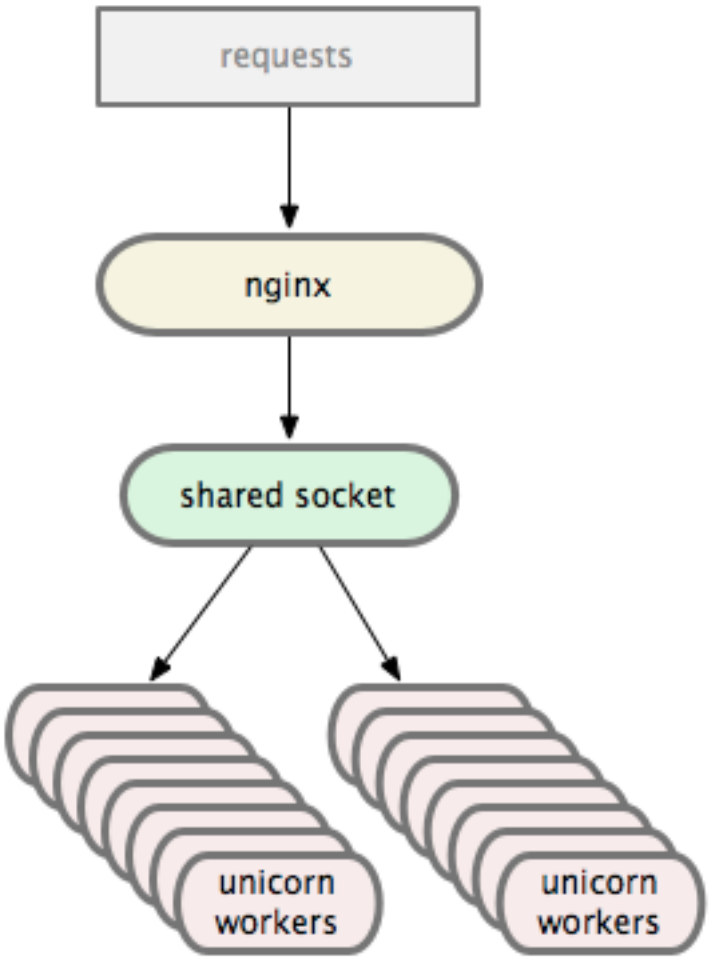
Server Architecture

The production application server needs to be able to handle HTTP requests. To do this, nginx [6], a lightweight HTTP server, is employed to listen on port 80 for HTTP requests.

nginx will serve up static, compiled assets when requested out of the ^/assets directory. Any other requests are routed through an nginx proxy to Unicorn [11], an application server running the application level code.

A separate server is used to run MongoDB and Redis instances. Because both database implementations require both bus time, high memory usage, and many disk reads and writes, it makes sense from a performance standpoint to put them on their own servers. Scaling horizontally is also easier, as application servers can be imaged, spawned, and put behind a load balancer. Scaling database servers isn't as easy, but a similar process is used to shard and offset database requests once hardware limitations are reached.

The following is a simple diagram the path HTTP requests take from the request stage to the Unicorn workers, which are running the RoR application in the production environment.



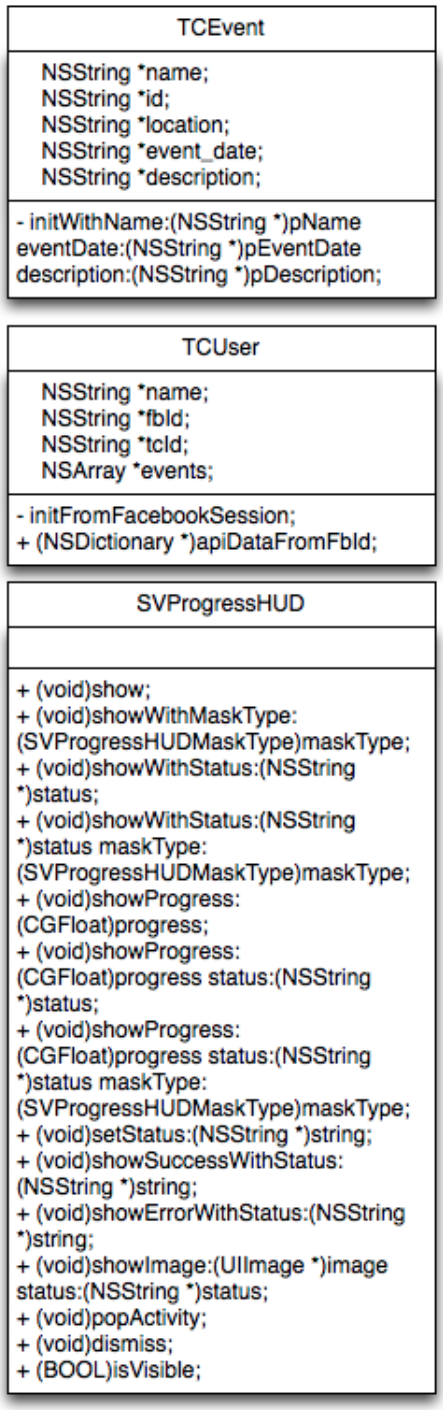
(diagram from https://github.com/blog/517-unicorn)

Design and Implementation of Mobile Application

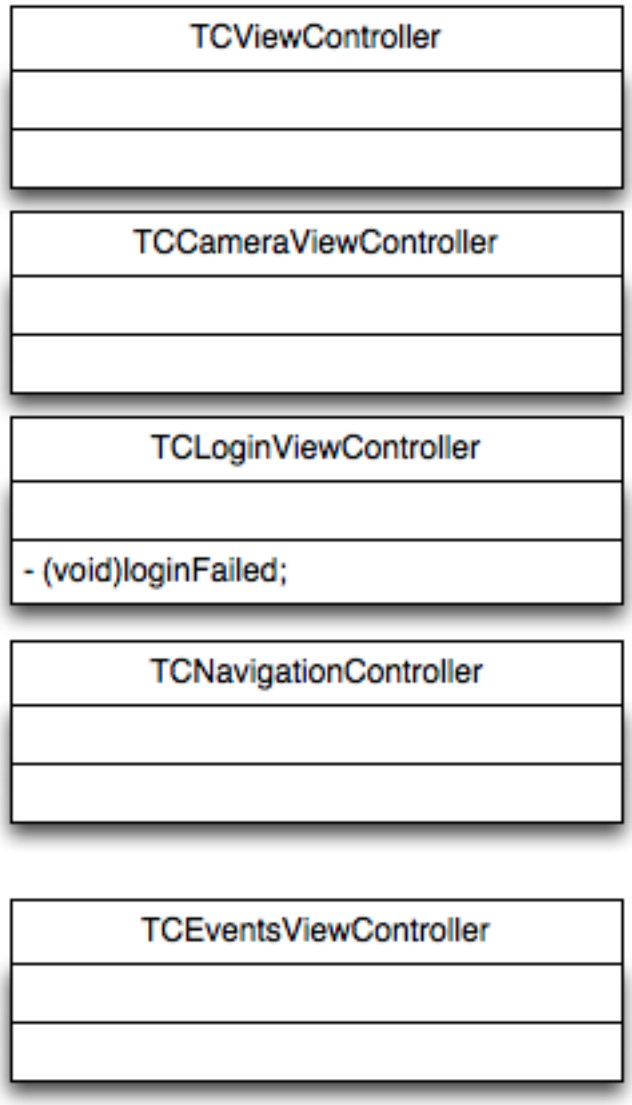
Vendor Application

The following diagrams are a simple visual model of both the models and view controllers for the iOS Vendor application, extracted from the Xcode project.

Models



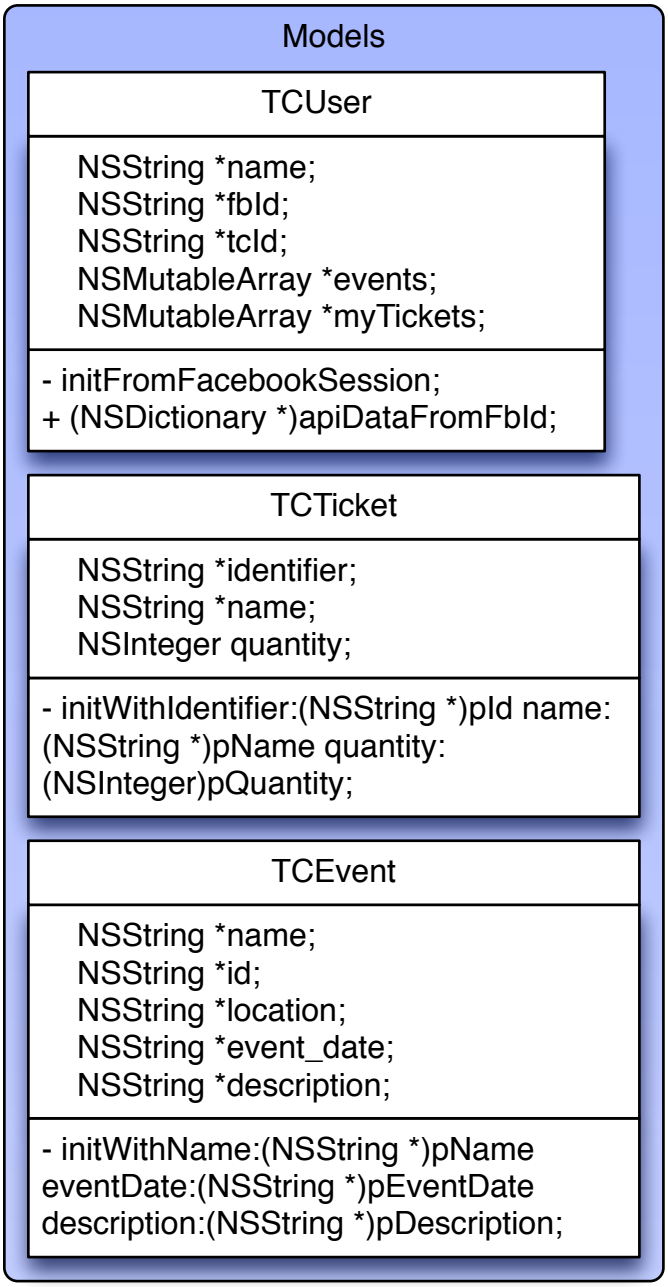
View Controllers



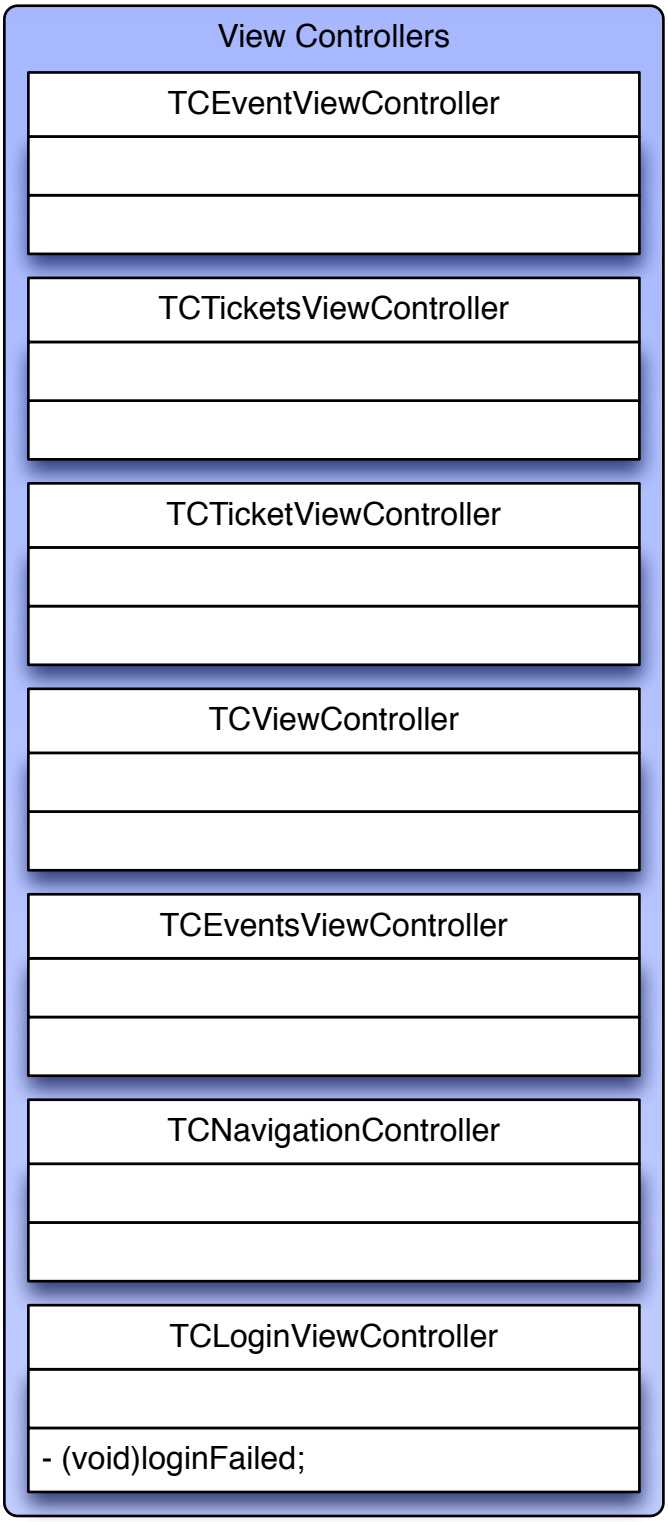
User Application

The following diagrams are a simple visual model of both the models and view controllers for the iOS User application, extracted from the Xcode project.

Models


Models
TCUser
NSString *name;
NSString *fbId;
NSString *tcId;
NSMutableArray *events;
NSMutableArray *myTickets;
- initFromFacebookSession;
+ (NSDictionary *)apiDataFromFbId;
TCTicket
NSString *identifier;
NSString *name;
NSInteger quantity;
- initWithIdentifier:(NSString *)pId name:
(NSString *)pName quantity:
(NSInteger)pQuantity;
TCEvent
NSString *name;
NSString *id;
NSString *location;
NSString *event_date;
NSString *description;
- initWithName:(NSString *)pName
eventDate:(NSString *)pEventDate
description:(NSString *)pDescription;

View Controllers



Testing

API testing was conducted using RSpec [9], specifically the rspec-rails gem. RSpec uses a ruby like syntax to define tests. A sample test that ensures a controller always returns a HTTP 200 response (success) is below:

```
require 'spec_helper'

describe EventController do

  describe "POST 'create'" do
    it "returns http success" do
      get 'create'
      response.should be_success
    end
  end

  describe "GET 'edit'" do
    it "returns http success" do
      get 'edit'
      response.should be_success
    end
  end

  describe "PUT 'update'" do
    it "returns http success" do
      get 'update'
      response.should be_success
    end
  end

  describe "DELETE 'destroy'" do
    it "returns http success" do
      get 'destroy'
      response.should be_success
    end
  end

  describe "GET 'show'" do
    it "returns http success" do
      get 'show'
      response.should be_success
    end
  end

end
```

Related Work

The focus of this related work analysis will be on the closest-in-similarity competitor, EventBrite. EventBrite, an established digital and physical ticket platform provides many of the same features as TicketCloud, with the exception of the Return-to-Market functionality for purchased tickets.

EventBrite is easier to use than its major market competitor, TicketMaster, but has a more cluttered user interface than TicketCloud's minimalistic design. However, EventBrite does allow a user to explore upcoming events by location – a user can easily find events within reasonable travelling distances.

Based on market share, another related service is TicketMaster, which typically powers many high profile events. TicketMaster recently released digital tickets, and the ability to transfer digital tickets between friends and family. While similar to TicketCloud's Return-to-Market feature, this does give the user the opportunity to "resell" their tickets to any of TicketCloud's customers.

Both competitors have mobile applications in the Apple App Store.

Conclusions and Future Work

The process from idea to mockups to product with customer development along the way over the two quarters was helpful and engaging. The development of a REST API, Website, and two iPhone applications came after a process of mockups, market surveys, development, and customer testing.

Before the TicketCloud platform can be brought to market a few action items need to be fulfilled. These include implementing a payment system, market viability analysis, as well as customer acquisition for both vendors and event goers. In order to complete these, the TicketCloud team would need to be expanded to include a business side that could conduct the majority of this research while the product is refined.

Future work includes Android application(s) to match the iPhone applications, ensuring that TicketCloud is usable across both major mobile platforms.

References

Num	Reference	URL
1	CoffeeScript	http://coffeescript.org/
2	Compass	http://compass-style.org/
3	HAML	http://haml.info/
4	MongoDB	http://www.mongodb.org/
5	Mongoid	http://mongoid.org/
6	nginx	http://nginx.org/
7	Ruby on Rails	http://rubyonrails.org/
8	Redis	http://redis.io/
9	RSpec	http://rspec.info/
10	SASS	http://sass-lang.com/
11	Unicorn	http://unicorn.bogomips.org/